Diseño e implementación de un generador de código para modelos de interfaces gráficas en el lenguaje ActionGUI

MEMORIA DE SISTEMAS INFORMÁTICOS

Gonzalo Ortiz Jaureguizar

Departamento de Sistemas Informáticos y Programación
Facultad de Informática
Universidad Complutense de Madrid

Junio 2011

Documento maquetado con TEXIS v.1.0.

Diseño e implementación de un generador de código para modelos de interfaces gráficas en el lenguaje ActionGUI

Memoria de Sistemas Informáticos

Dirigida por el Doctor
Manuel García Clavel

Departamento de Sistemas Informáticos y Programación
Facultad de Informática
Universidad Complutense de Madrid

Junio 2011

Resumen

As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming had become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them, it has created the problem of using its products. Edger W. Dijkstra, EWD 340: The humble programmer. ACM 15 (1972), 10 : 859-866

La programación de aplicaciones no ha cambiado sustancialmente desde que los lenguajes de alto nivel hicieron acto de presencia. Muchos pensaron en los años noventa que el desarrollo dirigido por modelos proporcionaría las herramientas necesarias para hacer del desarrollo de software una verdadera ingeniería, pero lo cierto es que no ha llegado a penetrar realmente en una industria donde hasta que la aplicación no ha sido al menos parcialmente codificada es difícil saber si cumple o no los requisitos impuestos por el cliente.

El santo grial que la arquitectura dirigida por modelos promete es la generación de código automática, esto es: Los ingenieros diseñan el modelo de la aplicación y tras unas transformaciones automáticas, el código de la aplicación se genera automáticamente. Si bien para aplicaciones de ámbito general esto se encuentra lejos de la realidad, han ido surgiendo herramientas capaces de traducir modelos a aplicaciones en ámbitos concretos. Un ejemplo de estas herramientas es SmartGUI, que mediante un lenguaje de modelado propio es capaz de generar la aplicación garantizando además la seguridad de esta. Para ello la herramienta se orienta al desarrollo de aplicaciones cuyo objetivo sea trabajar con seguridad sobre un modelo de datos.

Este texto describe primero los lenguajes usados por SmartGUI, cómo se describen los datos de la aplicación, las normas que rigen quién puede acceder a ellos y en qué modo, cómo diseñar el flujo del programa a traves de ventanas y el lenguaje en el que sintetiza estas tres facetas de la aplicación. A continuación se trata cómo se traducen estos lenguajes en código computable y finalmente estudia SmartGUI Viewer, el motor que ejecuta la aplicación generada interpretando el propio modelo, explicando cómo se transladan conceptos abstractos de SmartGUI a entidades concretas.

Índice

Índice de figuras

Capítulo 1

Introducción

Resumen: Este capitulo está dedicado a definir los conceptos en los que se basa SmartGUI. Comienza explicando la arquitectura dirigida por modelos, base sobre la cual se construye la herramienta. Continúa describiendo las aplicaciones que SmartGUI permite modelar, las llamadas aplicaciones de gestión de datos. Finalmente, puesto que SmartGUI Viewer permite ejecutar las aplicaciones modeladas con SmartGUI como aplicaciones web, se dedica una sección a analizar algunas de las tecnologías usadas en este campo.

1.1. Arquitectura dirigida por modelos

La arquitectura dirigida por modelos o MDA (Object Management Group, 2003) especifica un método de desarrollo de software mediante modelos. Tal y como se usarán en este texto, un modelo es al universo que modela como un diagrama de clases al diagrama de objetos derivado, esto es, cualquier objeto válido del universo modelado tiene que poder representarse con una instancia del modelo.

En la metodología MDA el sistema ha de definirse en primer lugar con un modelo independiente de plataforma (*Platform Independient Model* o PIM), es decir, lo bastante genérico para especificar el sistema sin hacer uso de ninguna plataforma tecnológica[1] concreta. Este sistema se querrá implementar en una o varias plataformas tecnológicas, MDA dice que para cada una de estas tecnologías se ha de definir un modelo dependiente de plataforma (*Platform Definition Model* o PDM) propio y para cada PDM, una transformación que tomando una instancia del modelo PIM como entrada genere una instancia del modelo PDM.

[1]Entendiendo como tal lenguajes de programación, sistemas operativos, bases de datos...

Teniendo estas transformaciones, modelando una única vez el sistema en PIM este puede traducirse automáticamente a varios modelos dependientes de plataforma. Si estos modelos PDM son suficientemente cercanos a la plataforma tecnológica que modelan, la traducción del modelo a código ha de ser sencilla, lo cual implicaría que, definiendo una única vez las transformaciones, podríamos generar el código no solo de una aplicación, sino de cualquier aplicación modelable por PIM de manera automática. Incluso si las corrección de la transformación estuviera demostrada, se podría no solo generar la aplicación automáticamente sino que, además, se podría garantizar que esta es tan correcta como lo era en el modelo abstracto de partida.

Si el modelo PIM es comprensible por el cliente que solicita la aplicación, los requisitos de la misma pueden plasmarse diréctamente en él, sin tener que recurrir a otro modelo de documentación. En un caso ideal esto significaría no tener que esperar a tener una versión de prueba del producto para saber si es conforme o no a lo solicitado por el cliente.

Como puede verse, los parrafos anteriores están repletos de condicionales y de suposiciones. Si todas se cumplieran esta metodología sería la única usada en la industria y numerosos programadores estarían sin empleo. Por suerte o por desgracia las condiciones antes expuestas no son sencillas de cumplir. Definir un modelo independiente de plataforma lo suficientemente sencillo como para poder modelar sobre él de manera que sea comprensible por las personas y a la vez lo suficientemente potente como para especificar tan detalladamente el comportamiento del sistema como para poder generar código automáticamente no es en absoluto sencillo. Es aun más complicado si se pretende que este modelo sea generalista, esto es, que sirva para implementar cualquier tipo de aplicación.

Los lenguajes de programación modernos están fuertemente especificados, por lo que teóricamente no debiera ser difícil diseñar un modelo dependiente de plataforma para ellos. Aun así para poder generar código automático para una cierta plataforma habría que tener un modelo PDM para cada tecnología. Esto es, si quisieramos generar una aplicación Java con entorno de ventanas podría haber un modelo para *AWT*, otro para *Swing*, otro para *SWT* y otro para *JavaFX* y consiguientemente tendría que haber una transformación para cada uno de ellos.

Precisamente las transformaciones de modelos son otro talón de Aquiles de MDA. Para que estas sean independientes de plataforma y claras, OMG ha especificado lenguajes de transformación. QVT (Object Management Group, 2005) y MOF2Text son los lenguajes especificados respectivamente para realizar transformaciones de modelo a modelo y de modelo a texto (es decir, generar código). Sin embargo sólo existen implementaciones parciales de terceros de estos lenguajes cuya eficiencia y documentación no es comparable a la de lenguajes de programación más maduros y de carácter comercial. Por ejemplo QVTO (Eclipse Model to Model (M2M) Project, 2011) implementa

la parte operacional (imperativa) de QVT y Acceleo de Obeo, implementa MOF2Text.

1.2. Aplicaciones de gestión de datos

SmartGUI utiliza la metodología MDA, partiendo de varios lenguajes PIM muy concretos: uno para modelar los datos de la aplicación, otro para modelar la seguridad y otro para modelar el flujo de ventanas. Estos modelos se mezclan, mediante transformaciones QVT en un único modelo de aplicación y se delega en módulos externos el generar las aplicaciones a partir este.

Además SmartGUI está concebido para aplicaciones centradas en la gestión de datos de manera segura. Esto es, SmartGUI no pretende modelar aplicaciones de ámbito general sino sistemas más concretos, como puede ser la gestión de una intranet donde se realicen informes o se administren una serie de recursos, ganando especial interés si diferentes usuarios tienen diferentes permisos sobre estos recursos. Incluso así, los componentes gráficos suministrados por SmartGUI son limitados y estos recursos han de poder presentarse de manera sencilla mediante cadenas de texto. Por ejemplo, una aplicación GIS, donde la gestión de datos es importante pero estos se presentan habitualmente sobre un mapa, no tiene cabida actualmente en la herramienta.

Si bien se trata de un campo muy concreto de aplicaciones, estas son usadas en multitud de sectores. La tendencia actual en estas aplicaciones es desarrollar una aplicación que permita, por ejemplo, administrar los empleados y sueldos de una empresa, gestionar automáticamente el inventario de un almacén o gestionar el funcionamiento de un centro sanitario. Frente a estos sistemas, SmartGUI permite diseñar de manera rápida una aplicación a medida del cliente e incluso modificarla de manera sencilla para adaptarla a un nuevo requisito.

1.3. Tecnologías web

Las características de las aplicaciones web las hacen idóneas para implementar las aplicaciones generadas por SmartGUI. El hecho de que los datos se almacenen en un servidor externo a la aplicación cliente otorga mayor la seguridad a la gestión de los mismos y la simplicidad de los componentes gráficos de SmartGUI hace que estos sean fácilmente implementables código HTML. A esto hay que sumar otras ventajas de las aplicaciones web: las aplicaciones cliente estarán siempre actualizadas, serán accesibles desde cualquier lugar (siempre y cuando así se desee) y, además, gracias al uso de estándares, será multiplataforma.

Prueba del éxito de esta clase de aplicaciones es que existen multitud de tecnologías y frameworks para implementar aplicaciones web. Las primeras versiones de SmartGUI, por ejemplo, creaban aplicaciones implementadas en PHP que generaban páginas HTML planas en las cuales el usuario daba valor a los componentes gráficos y, al pulsar en el botón correspondiente, esta información era enviada al servidor.

Esta implementación es sencilla pero hace que la aplicación generada sea muy poco interactiva, por lo que resulta necesario añadir código JavaScript a las páginas generadas. Antes de añadir este código, se estudió las consecuencias que tendría, siendo las principales las siguientes;

- Cada navegador web tiene su propio intérprete JavaScript que en ciertas situaciones no cumple el estándar, por lo que es necesario hacer una implementación para cada navegador, aumentando así el coste de desarrollo de SmartGUI.
- Encontrar y solucionar errores JavaScript es una tarea compleja, pues precisamente por ser interpretado de distintas maneras por cada navegador, las herramientas para depurar la ejecución son pobres y escasas.
- En ciertas situaciones sería necesario, desde el cliente, recuperar información del servidor de base de datos, lo cual, con la implementación PHP imponía recargar la página, almacenando a la vez los valores introducidos por el usuario. Esto dificultaba la implementación del código.

Son muchas las empresas que desarrollan páginas web con la suficiente interactividad como para encontrar estos problemas, por lo que existen varios frameworks para desarrollar aplicaciones web que intentan solucionar esto de diferentes maneras. Por ejemplo, JQuery es una librería JavaScript que facilita la compatibilidad con múltiples navegadores, enmascarando, tras una función común, la presencia de una implementación para cada navegador. Sin embargo la depuración sigue siendo compleja en relación a lenguajes imperativos más maduros.

1.3.1. Google Web Toolkit

Conscientes de los problemas inherentes de JavaScript, Google lanzó Google Web Toolkit (GWT) en 2006. Como es habitual en el desarrollo de aplicaciones web, GWT impone diferenciar la parte de la aplicación que se ejecutará en el servidor y la parte que se ejecutará en el cliente. A diferencia de otros frameworks, GWT permite que ambas partes estén programadas en Java.

Durante el compilado, la parte del servidor sigue el proceso común de cualquier aplicación Java, generando el correspondiente Java bytecode que interpretará la máquina virtual Java del servidor. Sin embargo los navegadores web no son capaces de interpretar Java bytecode, sino HTML y

JavaScript. GWT primero compila las clases Java del cliente a bytecode y, partiendo de este código, genera páginas HTML con sus correspondientes funciones JavaScript. Si el proceso acabase aquí realmente no ofrecería grandes ventajas respecto a otros frameworks, pero la aplicación del lado del cliente generada cumple las siguientes propiedades:

1. Garantiza que el código JavaScript funcionará en cualquiera de los grandes navegadores web tanto de escritorio como de dispositivos móviles.

2. Garantiza que el código JavaScript generado será eficiente.

3. Tiene varios niveles de compilación, pasando de los más descriptivos a los más ofuscados. Los primeros son útiles mientras se desarrolla la aplicación para comprender qué tareas realiza cada script, mientras los segundos dificultan la lectura a un cliente malicioso y además reducen sustancialmente el número de caracteres de código generado, lo cual aumenta el rendimiento del ancho de banda del servidor.

4. Permite hacer llamadas al servidor mediante llamadas a procedimientos remotos (Remote Procedure Call o RPC). El servidor recibe una petición, realiza los cálculos pertinentes y devuelve la información. En concreto el servidor podría hacer consultas a la base de datos, lo cual resulta especialmente útil en SmartGUI.

5. Existen extensiones para añadirlo a cualquiera de los grandes entornos de desallo Java, prestando soporte oficial para Eclipse.

6. Aunque en la aplicación terminada la parte del cliente esté escrita en HTML y JavaScript, durante la fase de desarrollo la aplicación puede ser depurada sobre el código Java, usando para ello las herramientas que el programador considere más oportunas incluyendo la depuración paso a paso.

Además, al igual que AWT y Swing, las librerías gráficas de GWT están basadas en componentes que disparan eventos que a su vez son atentidos por oyentes, por lo que desarrollar aplicaciones web con GWT resulta familiar a un desarrollador de aplicaciones de escritorio.

Una prueba del éxito de GWT son los frameworks de terceros que se basan en él. Algunos de ellos extienden GWT con nuevos componentes, como en el caso de SmartGWT. Otros, como Vaadin, añaden una nueva capa de abstracción.

1.3.2. Vaadin

Vaadin es un framework de desarrollo de aplicaciones web que, al contrario que GWT, se caracteriza porque toda la lógica de la aplicación está en

el servidor. En GWT, cada vez que del lado del cliente se produce un evento (pulsar una tecla, hacer un click, etc) el cliente realiza la lógica asociada y solo en caso de necesitarlo, recurre al servidor con una llamada RPC. En Vaadin el cliente no matiene la lógica de la aplicación, sino que únicamente sabe qué componentes gráficos se muestran y cómo comunicar los eventos al servidor. El servidor, por otra parte, almacena toda la lógica y una copia del estado de la interfaz gráfica del usuario. Cuando se produce un evento, el cliente notifica al servidor que este se ha producido, el servidor ejecuta la lógica, modifica el estado de visualización de la aplicación y se lo comunica al cliente, que actualiza su estado si es necesario.

El modelo centrado en el cliente que usa GWT tiene las ventajas de consumir un menor ancho de banda y que la carga computacional del servidor es menor, puesto que delega gran parte del cálculo en el cliente. Por otro lado el modelo centrado en el servidor facilita la implementación de una política de seguridad y, de cara al programador, no es necesario preocuparse en absoluto de la sincronía entre el cliente y el servidor, pudiendo ver el sistema como una aplicación que se ejecutará en una misma máquina.

Las versiones modernas de Vaadin usan GWT en el lado del cliente. Por cada componente Vaadin en el servidor existe un componente en el lado del cliente. Estos componentes de cliente tienen un núcleo GWT y una capa Vaadin que se encarga de implementar la sincronización entre el estado en el cliente y el servidor.

1.3.3. Limitaciones de las tecnologías web

Gracias a frameworks como los aquí mencionados y a nuevas tecnologías y protocolos, las aplicaciones web se encuentran muy cercanas a las aplicaciones de escritorio. Sin embargo todavía existen ciertas limitaciones impuestas por la propia arquitectura cliente-servidor en la que estas aplicaciones se basan.

El caso más claro es el de las llamadas "noticiaciones push" o eventos disparados desde el servidor. La arquitectura cliente-servidor impone que sea el cliente el que realice una petición y que sea el servidor el que la conteste. Esto es debido a que en esta arquitectura, un cliente es aquel que hace peticiones y nunca las espera. Sin embargo no es raro que en una aplicación se produzca un cambio de estado sin que intervenga el cliente. Por ejemplo, en una aplicación de chat en la cual se comunican dos clientes (cliente A y cliente B) a través de un servidor, el funcionamiento más intuitivo sería que al recibir un mensaje del cliente A, el servidor enviase el mensaje al cliente B. Sin embargo en la práctica, hasta que el cliente B no comienza una nueva petición, el servidor no puede comunicarle el cambio. Las aplicaciones web que se enfrentan a estas situaciones suelen solventarlas de una de estas dos maneras:

- Mediante *polling*, es decir el cliente hace uso de un temporizador que, cada cierto tiempo, hace peticiones al servidor, que contesta con los cambios que se hayan producido. En general estas peticiones y sus respuestas son vacías, lo cual sobrecarga la red de manera innecesaria.
- Técnicas artificiales como posponer la respuesta del servidor hasta que o bien pase un tiempo o bien se tenga información, lo cual simula que el cliente se encuentra esperando un mensaje.

Debido al auge de las aplicaciones web estas situaciones son cada vez más frecuentes, por lo que se están desarrollando tecnologías para implementar las notificaciones push, como por ejemplo WebSockets.

Capítulo 2

SmartGUI

Resumen: En este capítulo se explicarán los lenguajes de modelado que usa SmartGUI para el desarrollo dirigido por modelos de aplicaciones de interfaz gráfica segura. Estos lenguajes son: ComponentUML, para modelar los datos; SecureUML, para modelar la seguridad; ActionGUI, que modela los eventos gráficos; XYLayout, que modela la representación de los componentes gráficos; y SecumlAndGui, en el que la seguridad definida en SecureUML se refleja en los componentes gráficos.

Para mostrar los conceptos principales de modelado se usará el ejemplo de una aplicación llamada ChitChat, que soporta múltiples salas donde los usuarios pueden conversar.

SmartGUI hace uso de la metodología MDA para generar aplicaciones centradas en los datos cuyo objetivo sea manipular de manera segura información presentada de una manera concreta, evitando así los problemas que ocasiona el generar aplicaciones de carácter general. A cambio, además de generar la aplicación automáticamente a partir de modelos, la aplicación generada cumplirá la seguridad impuesta por el modelador. Puesto que se trata de un proyecto académico, los lenguajes que define se mantienen simples para facilitar su implementación no comercial, por lo que no ofrece al desarrollador la capacidad usar servicios externos (como aplicaciones o llamadas al sistema operativo de la máquina que las ejecute), modificar dinámicamente ninguna faceta de la propia aplicación (ni el esquema de datos, ni la seguridad, ni el flujo de ventanas, ni el aspecto de los componentes gráficos) en ejecución, no es extensible y suministra un conjunto de componentes gráficos bastante escaso en comparación con otros framewoks.

2.1. OCL, restricciones y consultas

Object Constraint Language u OCL (Object Management Group, 1997) es un lenguaje diseñado para realizar consultas e imponer restricciones us-

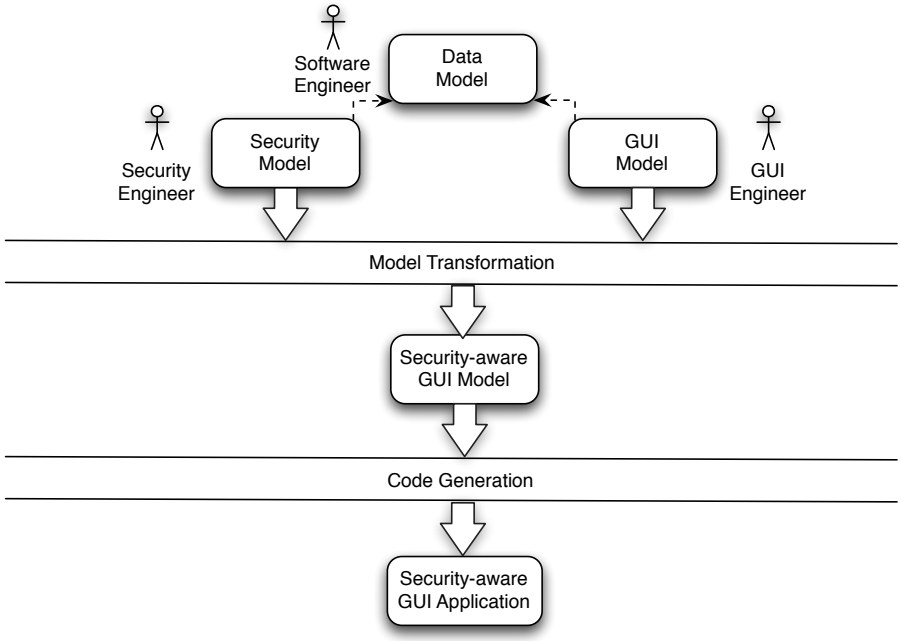

Figura 2.1: Desarrollo dirigido por modelos usado en SmartGUI.

ando notación textual. Como parte del estándar UML, fue originalmente diseñado para especificar propiedades que no podían ser capturadas de manera sencilla o natural usando notaciones gráficas (por ejemplo, invariantes en un diagrama de clases UML). Por ello las expresiones OCL se escriben en el contexto de un modelo y son evaluadas sobre una instancia de ese modelo. Las expresiones OCL son *side-effect free*, es decir, mantienen inmutable la instancia sobre la que se evalúan. Además se trata de un lenguaje fuertemente tipado, lo cual implica que todas sus expresiones han de tener siempre un tipo (ya sea primitivo, una clase o una colección).

Como lenguaje de restricciones, OCL se usa para especificar los estados válidos del modelo sobre el que versa, siendo válidos solo los estados que cumple una cierta expresión OCL booleana. Como lenguaje de consultas, OCL es usado para analizar y obtener información de la instancia sobre la que se evalúa.

Aunque la especificación OCL es reducida en comparación con otros lenguajes, es demasiado extensa como para resumirla en este texto, por lo que solo se enumerarán a modo de ejemplo algunas de las operaciones especificadas:

1. **includes**: Comprueba si un objeto es parte de una colección.

2. **isEmpty**: Comprueba si una colección es o no vacía.

3. **operador "."**: Que dado un objeto permite acceder a un atributo o asociación. Por ejemplo obj1.att1 devuelve el valor de atributo de nombre *att1* del objeto *obj1*.

4. **allInstances**: Que dada una clase, devuelve un conjunto de todos los elementos de esa clase presentes en la instancia donde la expresión se evalúa.

5. **forAll**: Que dado un conjunto chequea que todos sus elementos cumplan una cierta condición. Por ejemplo, si *integerSet* es un conjunto de enteros, la expresión integerSet−>forAll(i | i > 0) será cierta si todos los números de esa colección son mayores que 0.

6. **exists**: Que dado un conjunto chequea que alguno de sus elementos cumplan una cierta condición. Por ejemplo, si *integerSet* es un conjunto de enteros, la expresión integerSet−>exists(i | i > 0) será cierta al menos uno de los números de esa colección es mayor que 0.

7. **select**: Que dado un conjunto, devuelve todos los elementos de esa colección que cumplan una cierta condición. Por ejemplo, si *integerSet* es un conjunto de enteros, la expresión integerSet−>select(i | i > 0) devolverá el conjunto de números que sean mayores que cero.

2.2. Metamodelos

Siguiendo los principios MDA, SmartGUI define una serie de modelos independientes de plataforma.

2.2.1. ComponentUML: modelo de datos

ComponentUML es el lenguaje que SmartGUI usa para modelar los datos de la aplicación. Además sirve como el contexto para el lenguaje OCL, que se usará tanto para definir restricciones de seguridad como para realizar consultas en la interfaz gráfica.

El lenguaje proporciona una vista orientada a objetos de los datos del sistema, especificando su estructura y sus relaciones. Esencialmente ComponentUML es un subconjunto de los diagramas de clase UML donde las entidades (clases) se relacionan mediante asociaciones y poseen atributos. Tanto atributos como asociaciones tienen un tipo, que en el caso de los primeros ha de ser un tipo primitivo (entero, real, string, booleano...) y en el de las asociaciones es la entidad destino con la que enlaza su origen.

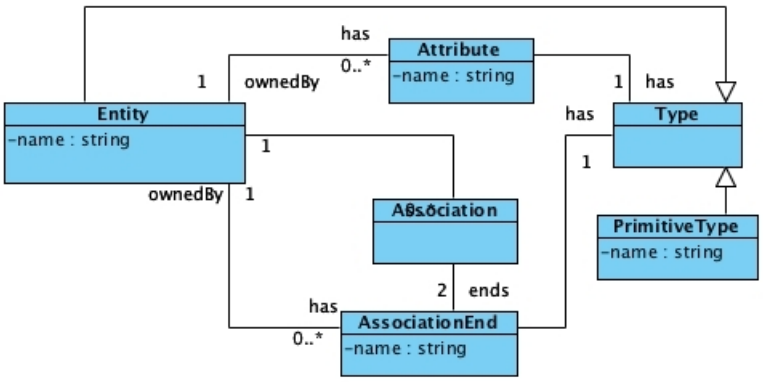

Figura 2.2: El metamodelo de ComponentUML.

Modelo de datos de ChitChat

En la aplicación de chat que hace de ejemplo existen usuarios, salas y mensajes. Cada usuario tiene un nombre de usuario, una contraseña, una dirección de correo electrónico, un mensaje de estado ánimo y un estado de conexión. Además puede participar o ser invitado a participar en un número indeterminado de salas de chat. Cada una de estas salas ha sido creada por un usuario y posee una fecha de creación y otra de finalización y gestiona los mensajes enviados por los usuarios que participan en ella.

La figura 2.3 especifica el modelo de datos de la aplicación, siendo una instancia del metamodelo ComponentUML

Como ejemplo de la sintaxis de OCL, se escriben algunas de las invariantes que el sistema debería cumplir. Por ejemplo, la siguiente expresión formaliza que los nombres de usuario han de ser unívocos, algo que únicamente con el diagrama sería imposible de especificar:

```
ChatUser.allInstances()->forall(
    u1,u2| u1 <> u2 implies u1.nickname <> u2.nickname).
```

De la misma forma puede decirse que los estados en los que puede estar un usuario son *online* u *offline*:

```
ChatUser.allInstances()->forall(
    u|u.status='off-line' or u.status='on-line').
```

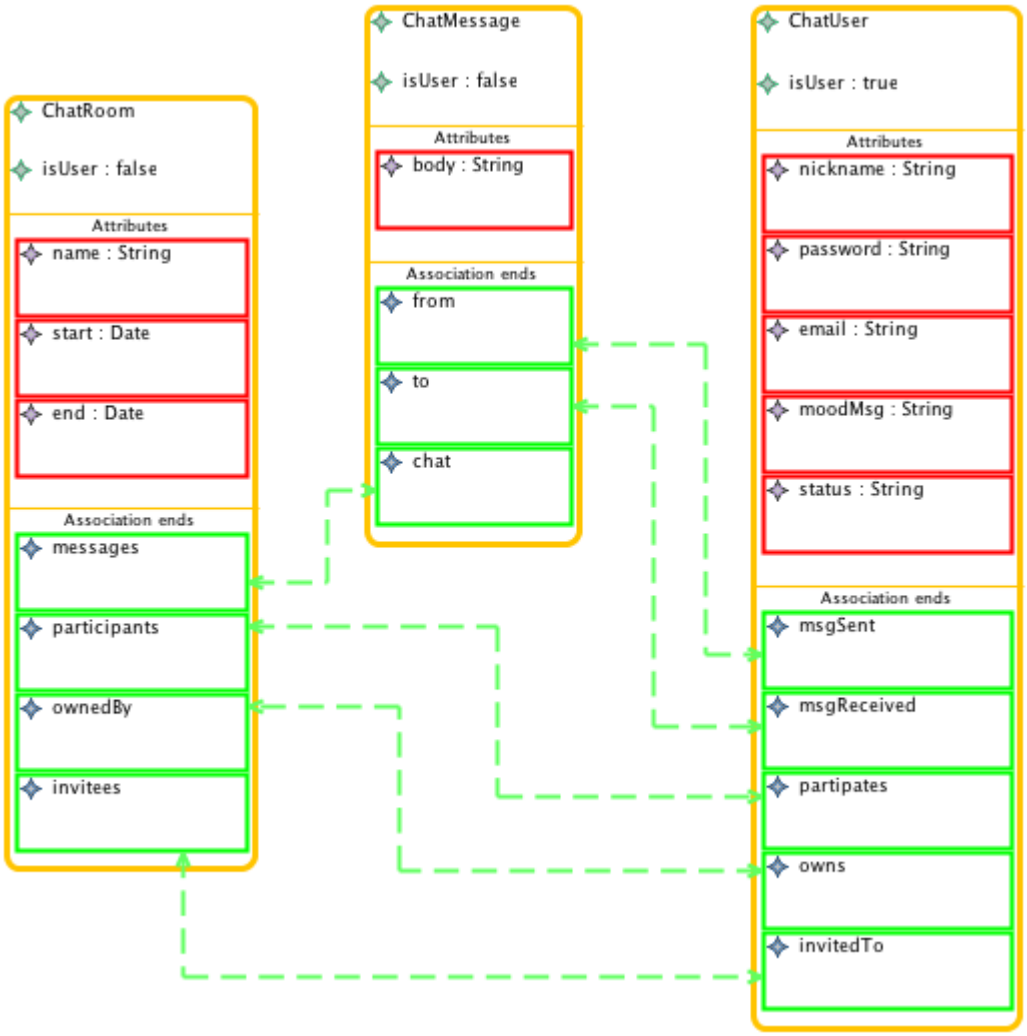

Figura 2.3: El modelo de los datos de ChitChat

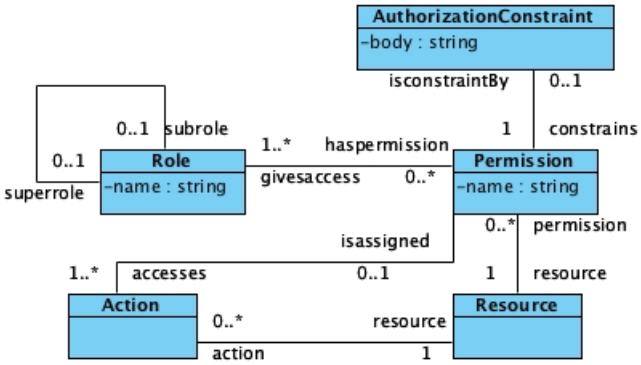

Figura 2.4: El metamodelo de SecureUML.

2.2.2. SecureUML+ComponentUML: Modelo de seguridad

SecureUML

SecureUML (Basin, Doser y Lodderstedt, 2006) es un lenguaje que permite formalizar el control de acceso a datos basado en roles[1]. Esto es, el lenguaje permite declarar unos *permisos* que especifican que *roles* tienen permiso para realizar ciertas *operaciones* sobre ciertos *recursos*. De esta manera según las responsabilidades y competencias de un usuario en la organización encargada de la aplicación, los usuarios tendrán un rol u otro.

Además, RBAC (y por tanto SecureUML) permite organizar los roles mediante herencia, donde unos roles hereden permisos de otros, lo cual permite representar una jerarquía de responsabilidades.

Sin embargo RBAC no permite especificar condiciones dinámicas a esos permisos. Por ello SecureUML extiende RBAC permitiendo asignar a los permisos una condición de acceso, es decir, una expresión booleana que se evaluará sobre el estado concreto del sistema cada vez que se intente acceder realizar esa operación sobre el recurso.

En resumen, en SecureUML a un rol se le otorgan todos los permisos asignados explícitamente y todos los que posean los roles de los que herede. Dada una operación sobre un recurso, un rol puede hacer uso de ella si y solo si posee algún permiso que le conceda el acceso a esa operación para el cual cumpla las restricciones dinámicas.

[1] Abreviado como RBAC (Ferraiolo, Sandhu, Gavrila, Kuhn y Chandramouli, 2001) por las siglas en inglés de Role-Based Access Control.

SecureUML+ComponentUML

SecureUML proporciona un lenguaje capaz de describir la política de control de acceso a acciones que actuan sobre recursos protegidos. Sin embargo delega en posibles "dialectos" el definir qué son los recursos protegidos y qué acciones pueden realizarse sobre ellos. Para modelar la política de control de acceso a los datos de la aplicación a generar, SmartGUI hace uso de un dialecto propio de SecureUML llamado SecureUML+ComponentUML. El metamodelo de este dialecto enlaza SecureUML con ComponentUML definiendo los recursos protegidos como las entidades, sus atributos y sus asociaciones y las acciones sobre ellos de la siguiente forma:

- **Atributos**
  - **lectura**: Leer el valor del atributo.
  - **actualización**: Actualizar el valor del atributo.
  - **acceso total**: Una acción compuesta que se traduce como la de lectura y actualización sobre el atributo.
- **Asociaciones**:
  - **lectura**: Leer el valor de la asociación.
  - **creación**: Añadir una nueva entidad a esta asociación.
  - **borrado**: Eliminar una entidad de esta asociación.
  - **acceso total**: Una acción compuesta que se traduce como la de lectura, creación y borrado sobre las asociación.
- **Entidades**:
  - **creación**: Crear entidades.
  - **borrado**: Borrar entidades.
  - **lectura**: Una acción compuesta que se traduce en la acción de lectura sobre cada uno de los atributos y las asociaciones de la entidad.
  - **actualización**: Una acción compuesta que se traduce en la acción de actualización sobre todos los atributos y la de creación y borrado de todas las asociaciones de la entidad.
  - **acceso total**: Una acción compuesta que se traduce como la de acceso total sobre todos los atributos y las asociaciones de la entidad

En SecureUML+ComponentUML las condiciones de acceso se definen mediante expresiones OCL a las cuales se le añaden cuatro palabras reservadas (self, caller, value, y target) con el siguiente significado:

- **self**: Se refiere al recurso (el atributo, la asociación o la entidad) sobre el que actuará la acción si se concede el permiso.
- **caller**: Se refiere al actor que ejecutará la acción si se concede el permiso.
- **value**: En el caso de la actualización de atributos, se refiere al valor al que se intenta actualizar. La versión presentada de SmartGUI Viewer no la soporta.
- **target**: En el caso de las creación o destrucción de asociaciones, se refiere a la entidad que se añadirá o se removerá de la misma. La versión presentada de SmartGUI Viewer no la soporta.

Modelo de seguridad de ChitChat

Un posible diseño de seguridad podría ser este[2]:

- Solo los administradores pueden crear y eliminar usuarios.
- Los administradores pueden leer el nombre de usuario, email, mensaje estado de ánimo y estatus de conexión de cualquier usuario.
- Cualquier usuario puede leer su propio usuario, email, mensaje de estado de ánimo y estatus de conexión.
- Cualquier usuario puede leer el nombre de usuario, estado de ánimo y estado de conexión de cualquier usuario.
- Los usuarios pueden unirse a una sala de chat solo por invitación, pero pueden abandonarla en cualquier momento.

La figura 2.5 muestra esta política de seguridad usando Secure-UML+ComponentUML.

2.2.3. ActionGUI: Modelo de eventos gráficos

ActionGUI es el lenguaje usado por SmartGUI para modelar la lógica de la interfaz gráfica, esto es, qué componentes gráficos existen y qué ocurre al interactuar con ellos, dejando de lado el cómo han de representarse estos componentes.

El metamodelo de ActionGUI, que puede verse en la figura 2.6, permite modelar una jerarquía de *widgets* (ventanas, campos de texto, botones, tablas...). Los widgets tienen asignadas *variables* y *eventos*. Las primeras almacenan información y los segundos son disparados durante el ciclo de vida

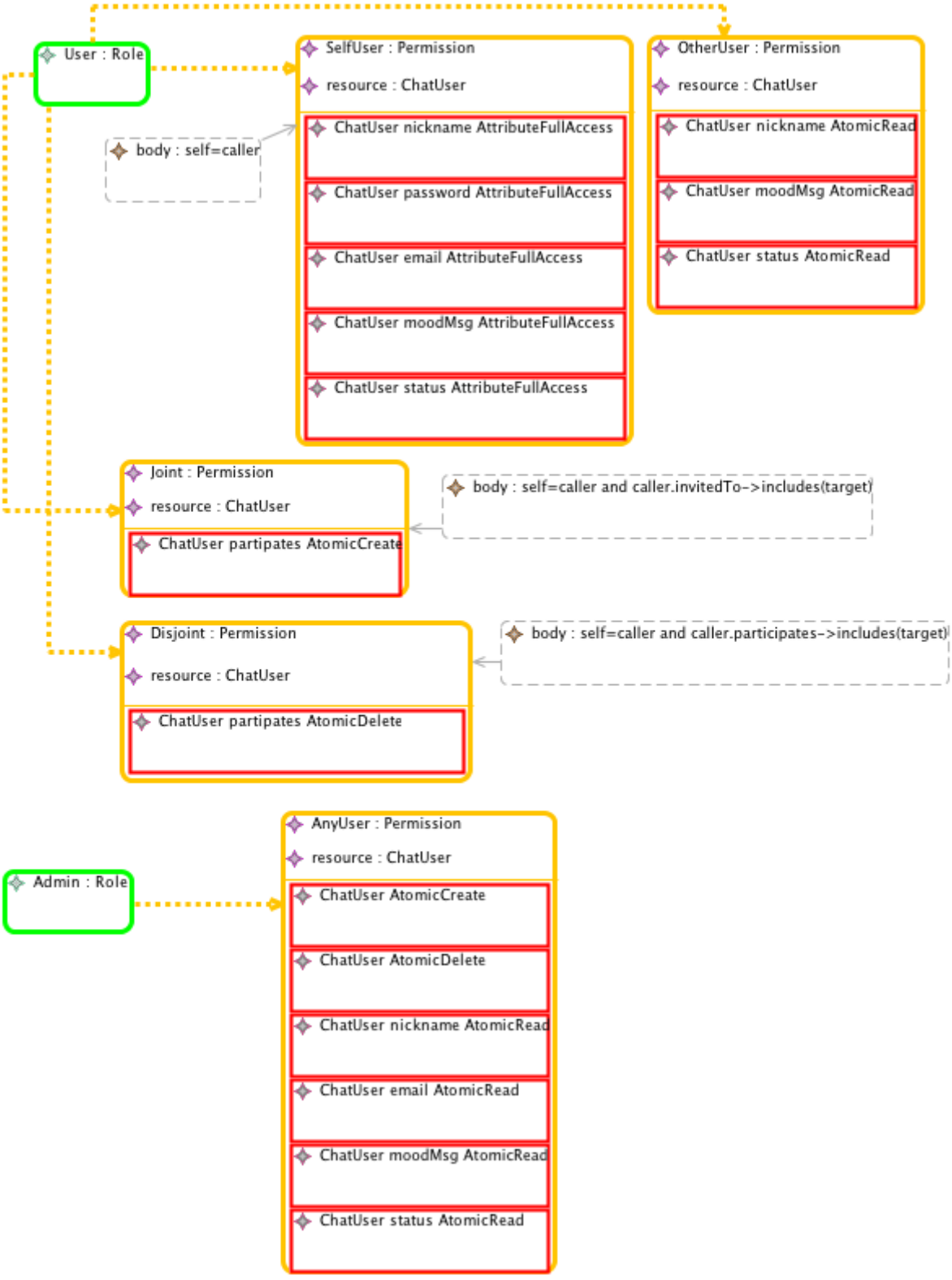

Figura 2.5: Ejemplo del control de acceso de la aplicación de chat.

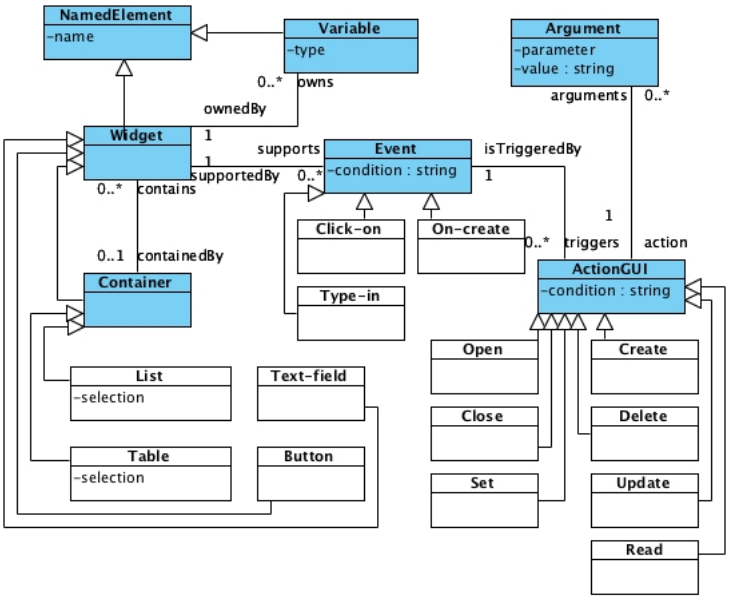

Figura 2.6: El metamodelo de ActionGUI.

del widget. Al dispararse uno de estos eventos, se ejecutan las *acciones* que este tiene asignadas.

Las acciones, su argumentos y ciertos atributos de los widgets (como la expresión selectora de las tablas y comboboxes) hacen uso de expresiones escritas en (OCL) extendido de manera que se permite el acceso a las variables de los widgets encerrandolas entre corchetes.

En ActionGUI todos los widgets tienen un *id* que les identifica unívocamente dentro de su contenedor. Además en el caso de las ventanas este identificador ha de ser unívoco también entre ellas. De esta manera puede fácilmente construirse para cada widget un identificador globalmente unívoco mediante el siguiente método recursivo:

```
Context Widget::idGlobal : String
    if self.container->isEmpty() then
        result := self.id
    else
        result := self.id.concat(self.container.idGlobal)
    endif
```

Este id global es el que se usa para acceder a una variable en las expre-

[2]Al no ser el objetivo de este texto, la política de seguridad mostrada pretende servir como un sencillo ejemplo, por lo que no es tan concienzuda como habría de serlo en una aplicación real

siones OCL. Si se quiere acceder a la variable aVar del widget aWidget que a su vez está contenido en aTable que finalmente está contenido en aWindow, habrá que escribir [aWindow.aTable.aWidget.aVar].

Widgets

ActionGUI soporta los siguientes widgets:

- **Windows**: Pueden contener cualquier tipo de widget excepto otras ventanas. Además no pueden ser contenidas por ningún otro widget. Tienen un atributo booleano *main* tal que Window.allInstances()->select(w | w.main)->size() = 1. La primera ventana que se abrirá al ejecutar la aplicación será aquella cuyo atributo main valga true.

- **Label**: Contiene una variable *text* de tipo String, cuyo valor se muestra en el widget.

- **Text field**: Es similar a un label pero permite escribir en él. El texto escrito modifica la variable *text*.

- **Button**: Es similar a un label pero además puede lanzar eventos *onClick*.

- **BooleanField**: Es similar a un button, pero posee además una variable booleana *value* que se muestra visualmente junto al *text*.

- **Table**: Es un contenedor de widgets que los distribuye en filas y columnas. Tiene un atributo *rows* que almacena una expresión que, al evaluarse, devolverá un conjunto de entidades del modelo de datos del mismo tipo. A cada columna de la tabla se le asocia uno de los widgets contenidos y a cada fila una de las entidades pertenecientes al resultado de evaluar la expresión. Posee además una variable *selected* en la que se almacenan las entidades asociadas a las filas que la tabla tenga seleccionadas.

- **ComboBox**: Son similares a las tablas pero solo pueden contener un único widget cuyo tipo, además, deberá ser un label. Se representa como un selector desplegable. Funcionalmente un objeto de clase *ComboBox* es similar a un objeto de clase *Table* de una única columna.

- **DateField**: Representa un calendario. Posee variables como *day*, *week*, *dayOfWeek*, *month* o *year* (entre otras) que ofrecen distintas vistas de la variable *date*, que almacena la fecha seleccionada en el widget. Cuando la fecha cambia, dispara eventos OnClick.

Tablas y comboboxes Las tablas y los comboboxes son widgets especiales en su forma de contener widgets. Mientras al diseñar una ventana se modelan los widgets que esta contendrá, en las tablas y comboboxes los widgets contenidos en el modelo hacen las veces de "plantilla" de los widgets que existirán cuando la aplicación se ejecute. En ejecución, al construir una tabla se instancia un widget mediante la plantilla correspondiente a su columna. Cada uno de esos widgets dinámicos está asociado a una entidad a través de la fila en la que se encuentra. El widget puede acceder a esa entidad mediante la *pseudo-variable*[3] *row* definida en la tabla.

Eventos y acciones

Los eventos definidos en el metamodelo de ActionGUI son los siguientes:

- **OnCreate**: Se dispara cuando se construye el widget.
- **OnClick**: Se dispara cuando se acciona el widget. Solo ciertos widgets como los botones, campos booleanos y campos de fecha los disparan.
- **OnDestroy**: Se dispara cuando se deja de mostrar el widget.

Estos eventos contienen acciones. Las acciones, a su vez puede ser compuestas, es decir, pueden contener otras acciones, formando así un árbol de acciones, de forma que una acción o bien está contenida por un evento o bien por otra acción. Al dispararse un evento se ejecutan las acciones recorriendo el árbol en profundidad. Las acciones que especifica ActionGUI son:

- **Create (entidades)**: Crea una entidad y una variable temporal, guardándo en ella la entidad. Toma dos argumentos:
  - **type**: El tipo de la entidad a crear.
  - **variable**: Identificador de la variable temporal a crear.
- **Delete (entidades)**: Elimina una entidad. Toma un argumento:
  - **object**: Una expresión OCL extendida que devuelve la entidad a eliminar.
- **Read**: Lee el valor de un cierto atributo de una entidad. Toma tres argumentos:
  - **attribute**: El atributo que se leerá.
  - **object**: Una expresión OCL extendida que devuelve la entidad de la que se leerá el atributo *attribute*.

[3]Se usa el término pseudo-variable porque aunque en el metamodelo se corresponda con una variable, su valor depende de quién sea el widget que la use, por lo que en un lenguaje de programación *clásico* se asemejaría más a un método

  - **variable**: El id global de la variable donde se depositará el valor del atributo.

- **Update**: Modifica el valor del atributo de una entidad. Toma tres argumentos:
  - **attribute**: El atributo que se modificará.
  - **object**: Una expresión OCL extendida que devuelve la entidad en la que se modificará el atributo *attribute*.
  - **value**: Una expresión OCL extendida cuyo valor se asignará al atributo.
- **Create (asociaciones)**: Asocia dos entidades mediante una asociación definida en el modelo de datos. Toma tres argumentos:
  1. **sourceObject**: El objeto origen de la asociación.
  2. **targetObject**: El objeto destino de la asociación.
  3. **associationEnd**: El nombre de la asociación por la cual se asociarán ambas entidades.
- **Delete (asociaciones)**: Elimina una asociación entre dos entidades. Toma tres argumentos:
  1. **sourceObject**: El objeto origen de la asociación.
  2. **targetObject**: El objeto destino de la asociación.
  3. **associationEnd**: El nombre de la asociación a eliminar.
- **Open**: Se trata de la acción principal de flujo de ventanas, pues sustituye la ventana actual por otra. Toma como argumento el id de la ventana a abrir y acepta opcionalmente por cada variable de la ventana a abrir, un argumento cuyo valor se asignará a la variable cuando la ventana se abra.
- **Back**: Cierra la ventana actual y vuelve a la anterior.
- **Exit**: Cierra la aplicación.
- **Set**: Modifica una variable de la ventana actual. Toma dos argumentos:
  1. **target**: el id global de la variable a modificar
  2. **value**: Una expresión OCL extendida cuyo valor se asignará a la variable.
- **Conditional**: Tiene como atributo una expresión OCL extendida de tipo booleano y dos grupos de acciones asociados al valor verdadero o falso de la expresión. Si la expresión se evalúa como cierta, se ejecutarán las acciones asociadas al valor verdadero, ejecutándose las otras en caso contrario.

Expresiones

Algunos widgets y acciones poseen argumentos que son expresiones OCL. Antes, se ha dicho informalmente que estas expresiones pueden contener, entre corchetes, referencias a variables. De manera más formal, estas expresiones OCL contienen otras subexpresiones que han de procesarse antes que la expresión OCL. Estas subexpresiones se evalúan en el contexto GUI de la ventana en la que se encuentran. Una vez que las variables son evaluadas (se conoce su valor y su tipo) puede evaluarse la expresión OCL de manera normal.

En el contexto de la ventana pueden referenciarse:

1. o bien variables de widgets.
2. o bien variables temporales generadas por acciones de creación de entidades que sean visibles dentro del ámbito de la acción que contenga la expresión a evaluar.

Variables de widget: Como se ha explicado antes, las variables de widget pueden referenciarse agregando su id al id global del widget que las contiene. Aunque de esta manera una variable puede distinguirse unívocamente de cualquier otra variable del modelo, al evaluarse en el contexto de una ventana, solo pueden accederse a las variables de los widgets de la ventana en la que se encuentran.

Variables temporales: Ciertas acciones, como las condicionales, contienen a su vez otras acciones, creando una jerarquía de acciones que es usada para definir los ámbitos de las variables temporales. Es decir, una acción siempre se encuentra contenida por o bien otra acción o bien por un evento (siendo estos la raíz del árbol de jerarquía de acciones). Así pues, si la acción *create1* está contenida por *contenedor1* e instancia la variable temporal *var*, esta variable será visible (es decir, podrá usarse) en cualquier acción *otra1* posterior a *creat1* siempre y cuando *otra1* sea contenida (directa o transitivamente) por *contenedor1*.

Flujo de datos entre ventanas: Al impedir referenciar variables de otras ventanas la única forma de transferir información entre ventanas es mediante los argumentos de la acción *open*. De esta manera la aplicación es más simple y los modelos son más fáciles de entender al no haber efectos laterales.

Modelo de flujo de eventos gráficos de ChitChat

Como ejemplo de flujo de ventanas se modela el proceso de acceso a la aplicación de chat.

La ventana inicial (*loginWi*) contiene:

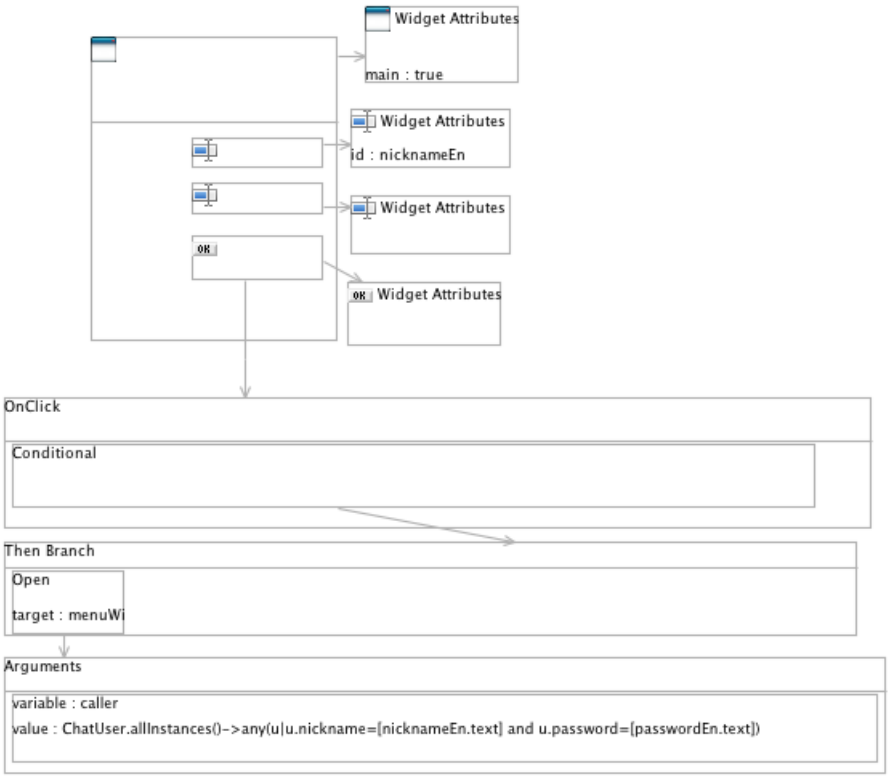

Figura 2.7: Modelo de la ventana inicial del chat.

- **nicknameEn**: Un text-field donde el usuario escribirá su nombre de usuario.
- **passwordEn**: Un text-field donde el usuario escribirá su contraseña de acceso.
- **loginBu**: El botón de acceso. Cuando el usuario lo pulse, se comprobará si existe algún usuario con el nombre de usuario nicknameEn.text cuya contraseña sea passwordEn.text. Si la autentificación tiene éxito el usuario será dirigido a una nueva ventana con un menú (*menuWi*) instanciando la variable *caller* con el usuario logeado.

2.2.4. XYLayout: Modelo de representación gráfica

Aunque ActionGUI defina el comportamiento de la aplicación generada, no especifica cómo debe mostrarse la aplicación. Lejos de ser un defecto, esto es una ventaja. Es una tendencia clara en las librerías gráficas el separar la funcionalidad de la aplicación de su representación. El ejemplo más claro es

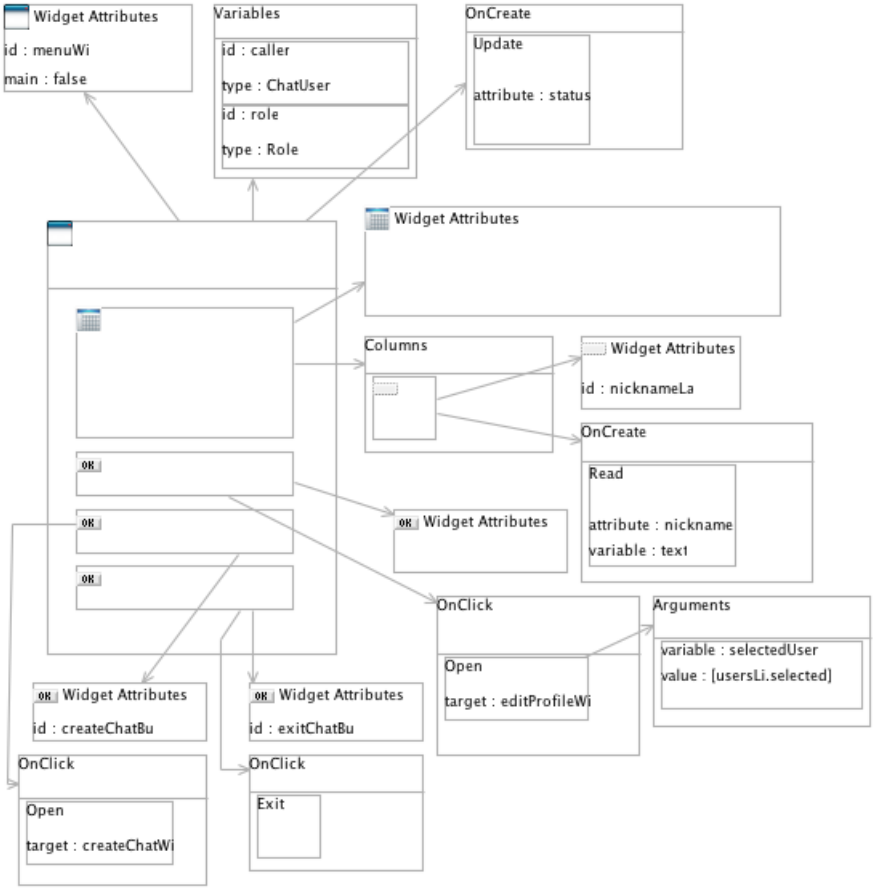

Figura 2.8: Modelo de la ventana "menú".

cómo se complementan HTML y CSS, pero también puede verse en otras entornos como las aplicaciones Android, GWT o JavaFx.

Gracias a esta separación entre la lógica y la presentación puede dividirse el trabajo entre distintas personas y una misma aplicación puede presentarse de la distintas maneras en distintos terminales (por ejemplo, dispositivos móviles y dispositivos de sobremesa) o para distintos públicos (por ejemplo personas con deficiencias visuales).

La presentación es uno de los puntos débiles de SmartGUI. Tanto es así que no especifica ningún modelo de representación gráfica. Actualmente, según como se hayan dibujado los widgets en el modelo ActionGUI, se crea un modelo sencillo cuyo metamodelo es XYLayout.

YXLayout es un lenguaje sencillo en el cual existen widgets que pueden contenerse entre sí y que tienen los siguientes atributos:

- **type**: tipo de widget en ActionGUI.
- **x**: coordenada x dentro de su contenedor.
- **y**: coordenada x dentro de su contenedor.
- **width**: anchura del widget.
- **height**: altura del widget.

2.2.5. SecumlAndGui: Modelo de aplicación

SecumlAndGui es un lenguaje de modelado resultado de la unión de ComponentUML, SecureUML+ComponentUML y ActionGUI. La función principal de SmartGUI es, partiendo de estos modelos, generar un único modelo SecumlAndGui en el que se especifiquen todos los aspectos de la aplicación.

El metamodelo de SecumlAndGui consiste en unir ComponentUML, ActionGUI y SecumlAndGui por la clase abstracta *Resource*. De ella heredan los eventos y las entidades y los *Resource* se asocian con acciones del modelo de seguridad. Las mayores diferencias con respecto a los metamodelos originales son las siguientes:

- Entre los permisos y las condiciones de acceso existe la clase *Formula*, de la que heredan *Conjuction*, *Disjuction* y *AuthorizationConstraint*. Las fórmulas, a su vez, pueden contener una serie de disjunciones o de conjunciones, lo que permite crear un árbol de restricciones de acceso. Estas asociaciones han de cumplir los siguientes invariantes:

  1.
```
Formula.allInstances()->forall(
      f | f.conjuction->notEmpty() implies (
          f.disjuction->isEmpty() and f.constraints->isEmpty()
```

```
        )
    )
```

2. ```
   Formula.allInstances()->forall(
       f | f.disjuction->notEmpty() implies (
           f.conjuction->isEmpty() and f.constraints->isEmpty()
       )
   )
   ```

3. ```
   Formula.allInstances()->forall(
       f | f.constaints->notEmpty() implies (
           f.conjuction->isEmpty() and f.disjuction->isEmpty()
       )
   )
   ```

- El conflicto de nombres entre las acciones de ActionGUI y las de SecureUML se ha resuelto cambiando el nombre de las acciones de ActionGUI a *GAction*.

- Las *GAction* pueden ser de tres tipos: Acciones sobre widgets (open, set, etc), acciones de flujo (condicionales) o *ModelAction*, es decir, acciones sobre el modelo de datos.

  - **WidgetAction**: Tienen un atributo enumerado *guiAction* que indica que tipo de acción sobre widgets se realiza y una referencia *actionOn* de tipo *Widget* que indica que widget modifica.
  - **ModelAction**: Tienen un atributo enumerado *modelAction* que indica que tipo de acción sobre el modelo de datos realiza y una referencia *subjet* a un objeto de clase *Subject*, que a su vez contiene un atributo de tipo String que será la variable temporal que instancie en el caso de las acciones de creación de entidades o una expresión OCL extendida en el resto de los casos.
  - **ConditionalActions**: Son similares a las acciones explicadas en ActionGUI.

- Las acciones de ActionGUI (*GAction*) dejan de tener los atributos como argumentos y pasan a tener asociaciones con objetos de clase *Parameter*.

- Los objetos de clase *Parameter* tienen una expresión OCL extendida y pueden opcionalmente instanciar una variable. El significado de parámetro es dependiente de la acción con la que esté asociado:

  - **Open**: Por cada variable de la ventana de destino que la acción modifique, existe un parámetro que instancia esa variable y cuya expresión es el valor a fijar.

  - **Set**: Tiene exáctamente un parámetro, que instancia la variable a modificar y cuya expresión es el valor a fijar.
  - **AssociationCreate**: Tiene exáctamente un parámetro que no instancia variables y cuya expresión marca la entidad a añadir como destino de la asociación.
  - **AssociationDelete**: Tiene exáctamente un parámetro que no instancia variables y cuya expresión marca la entidad a eliminar como destino de la asociación.
  - **AttributeUpdate**: Tiene exáctamente un parámetro que no instancia variables y cuya expresión marca el valor al que se fijará el atributo.

- Los eventos tienen una asociación con una colección de objetos de clase *Action* (acciones del metamodelo de seguridad) llamada *actions*. Un cierto rol podrá lanzar un evento si tiene permiso para ejecutar todas las *Action*s que este evento lance.

- Las tablas y comboboxes se traducen a *ComboBox*. Los objetos de esta clase se asocian con un objeto de clase *WidgetType*, que tiene como atributo una expresión OCL extendida cuyo significado es el mismo que el atributo rows de la clase *SelectableWidget* de ActionGUI.

2.3. Trabajo relacionado

Existen otras herramientas como WebRatio (Web Models Company, 2010), Olivanova (Care Technologies, 2011) y Lightswitch (Microsoft, 2010) que facilitan el desarrollo de aplicaciones centradas en los datos. En estas herramientas el proceso comienza definiendo un modelo de datos, sobre el cual se aplican diferentes patrones de generación de código que permiten crear, obtener, y modificar estos datos. A diferencia de la metodología usada por estas herramientas, SmartGUI ofrece mayor flexibilidad al usuario modelador al no limitarle con restricciones impuestas por un limitado número de patrones. Además estas herramientas imponen una cota de una o a lo sumo dos navegaciones dentre los datos.

Estas herramientas soportan las políticas RBAC de diferente manera. Lightswitch permite denegar o no a un cierto rol el permiso de ejecutar acciones así como de leer, actualizar o eliminar datos, pero el modelador tiene que programar manualmente estos comportamientos. WebRatio y Olivanova soportan también estos permisos y añaden la posibilidad de restringirlos bajo ciertas situaciones (de manera similar a las condiciones de acceso de SmartGUI) añadiendo precondiciones en las llamadas a las acciones que se realizan desde la GUI. Sin embargo ninguna de estas herramientas implementa el paso automático de la política de seguridad a la interfaz gráfica,

como sí hace SmartGUI.

Capítulo 3

Generación de código

Resumen: En el capítulo anterior se describieron los metamodelos mediante los cuales SmartGUI genera un único modelo SecumlAndGui. Este modelo es lo bastante genérico como para, partiendo de él, generar distintas aplicaciones cuya función y funcionamiento sería similar. En este capítulo se mostrará como, partiendo de este modelo, el módulo generador de código de SmartGUI es capaz de crear una aplicación ejecutable por SmartGUI Viewer.

En versiones de SmartGUI anteriores a la 2.0, la aplicación generada era una web PHP, de manera que se generaba un fichero por cada ventana que contenía un formulario que redirigía la aplicación a un nuevo fichero auxiliar asociado a esa ventana que, tras hacer las comprobaciones pertinentes, redirigía al navegador a la ventana a abrir. Además se generaban una serie de ficheros de configuración donde se almacenaban los métodos MySQL que debían llamarse para cada acción.

Las versiones modernas de SmartGUI generan aplicaciones web Java, por lo que el modelo de la aplicación puede leerse directamente de manera similar a la que SmartGUI utiliza en las transformaciones de modelo. Así, SmartGUI Viewer es en realidad una aplicación web que no depende del modelo SecumlAndGui, sino que interpretando este modelo, ejecuta la aplicación. Salvando las distancias, los modelos creados por el diseñador podrían equipararse al código fuente Java de una aplicación y el modelo SecumlAndGui al bytecode de la aplicación compilada. SmartGUI Viewer sería entonces la máquina virtual que ejecutaría la aplicación. Aun así, SmartGUI Viewer no acepta los modelos SecumlAndGui tal y como lo genera las transformaciones de modelo SmartGUI, siendo necesario realizar las siguientes tareas:

1. Traducir el modelo de datos a una base de datos SQL.

2. Traducir las expresiones OCL a procedimientos SQL.

3. Adaptar el modelo SecumlAndGui para saber a qué método SQL corresponde cada expresión OCL.

   Opcionalmente puede tomarse un modelo XYLayout y generar con él hojas de estilo CSS que SmartGUI Viewer usará para presentar interfaz. Ciertamente, el metamodelo SecumlAndGui podría almacenar también el propio diseño de la interfaz y en Vaadin, la tecnología en la que se basa SmartGUI Viewer, es posible fijar la posición y el tamaño de cada widget desde el código Java. Sin embargo es preferible que la presentación de la aplicación esté separada de la lógica para poder cambiar su aspecto a voluntad, por ejemplo, partiendo del mismo modelo SecumlAndGui, tener una versión para navegadores de escritorio y otra para sistemas con pantalla reducida, como los dispositivos móviles.

3.1. Traducción del modelo de datos a SQL

La transformación de ComponentUML a bases de datos relacionales es bastante simple. Las entidades se traducen a tablas donde cada fila es una entidad de ese tipo. Cada tabla tiene una columna para la clave principal y una columna por cada atributo de la entidad. Por cada asociación entre dos entidades, se crea en la base de datos una tabla con dos columnas: una que contiene la clave de la entidad origen y otra que contiene la clave de la entidad destino.

3.2. Traducción de OCL a SQL

Frente a la simplicidad de la traducción de ComponentUML a SQL, el transformar expresiones OCL a SQL es un proceso complejo a lo que hay que sumar el preprocesado de las expresiones GUI y de seguridad. Del preprocesado se encarga el propio módulo generador de código, mientras que traducir las expresiones OCL estándar a procedimientos SQL se delega en MySQL4OCL (Egea, Dania y Clavel, 2010).

Inicialización

Para traducir las expresiones, MySQL4OCL hace uso del analizador sintáctico de EOS (Clavel, Egea y de Dios, 2008), al cual hay que indicarle el modelo del contexto sobre el que se evalúan las expresiones, esto es, el modelo ComponentUML. Esta tarea es sencilla:

1. Las entidades de ComponentUML se traducen a clases en EOS.

2. Los atributos de ComponentUML se traducen a atributos en EOS.

3. Las asociaciones de ComponentUML se traducen a asociaciones en EOS.

Obtención de expresiones

Las clases que contienen expresiones son las siguientes:

- **ComboBox**: El objeto de clase *WidgetType* asociado posee una expresión *filter* que indica el número de filas del combobox y la entidad asociada a cada una de estas filas.
- **Parameter**: En su atributo *expression*.
- **ModelAction**: En su atributo *subject* salvo en el caso de las acciones de creación de entidades.
- **ConditionalAction**: En su atributo *condition*, que indica si se ejecutarán las acciones asociadas con la rama *then* o las asociadas con la rama *else*.
- **Event**: Las acciones de seguridad de su atributo *actions* tienen asociados uno o varios permisos que a su vez dependen de formulas. Las fórmulas de clase *BoolExpr* tienen asociada una expresión booleana que indica si se tiene o no permiso.

Para obtener las expresiones se recorre el modelo comenzando por las ventanas y descendiendo sucesivamente por los widgets contenidos. Para cada widget se recorren sus eventos, de estos se obtienen las acciones y de ellas los parámetros, analizando en cada objeto las expresiones OCL asociadas.

Preprocesado de subexpresiones

El preprocesado de las subexpresiones consiste en sustituir, en la expresión OCL, todas las subexpresiones encerradas entre corchetes por variables globales que han de declararse en EOS.

Las subexpresiones serán expresiones de seguridad si se encuentran en formulas o expresiones GUI en el resto de los casos. Las primeras pueden hacer referencia a la variable self como el recurso sobre el que actuará la acción asociada o a la variable caller como la entidad con la que se ha registrado el usuario, que por convenio estará asociado, de existir, a la variable homónima de la ventana activa. Las expresiones GUI pueden hacer referencia a variables de widget de la ventana activa (en cuyo caso se escribirá su id global) o, en el caso de las acciones y los parámetros, a variables temporales visibles, es decir, que hayan sido generadas por acciones ya ejecutadas cuyo contenedor contenga a la acción asociada a la expresión.

Al igual que las variables de un lenguaje de programación imperativo, las subexpresiones pueden clasificarse según su ámbito como globales o como locales. La subexpresion *caller* y casi todas las variables de ventana (todas menos la variable *row* de los objetos de clase *ComboBox*) se comportan como variables globales: son visibles desde cualquier expresión de una misma ventana y sus valores[1] no dependen del widget que la utilice. Las variables temporales, la subexpresión *self* y la variable *row* de los objetos de clase *ComboBox*, en cambio, se asemejan a las variables locales: dependiendo de en qué lugar se estén usando pueden ser visibles o no.

Para implementar el procesado de subexpresiones se usa una lista de tablas que relacionan una subexpresión con un tipo. Para resolver una subexpresión basta con recorrer la lista examinando si las sucesivas tablas contienen o no la subexpresión. Si ninguna tabla contiene la subexpresión, entonces esta no puede resolverse.

La tabla inicial se rellena con las variables de todos los widgets de una ventana (incluidas las propias variables de la ventana) a excepción de la variable *row* de los objetos de clase *ComboBox*, a los que solo puede accederse desde widgets[2] contenidos por el combobox.

Como se explicó con anterioridad, las expresiones OCL se buscan recorriendo un árbol que comienza en una ventana, recorre sus widgets y los eventos, las acciones y los parámetros. Este recorrido se hace en profundidad, lo cual permite añadir una tablas a la lista al comenzar a tratar un nodo, añadir a esta tabla las variables locales necesarias y eliminar esta tabla después de haber recorrido todos los hijos de este nodo. Este método, sin embargo, permite que cualquier expresión haga uso de varibles de widget y de la variable *caller*, permitiendo así que expresiones que por usar subexpresiones inválidas no son correctas, se traduzcan corréctamente. Puesto que SmartGUI chequea la corrección de las expresiones antes de generar el modelo SecumlAndGui no supone un problema siempre y cuando el modelo haya sido generado correctamente

A esta lista de tablas ha de añadirse niveles en los siguientes casos:

- En todos los objetos de clase *ComboBox*, añadiendo además la expresión *row*.

- En todos los eventos y todas las acciones condicionales, ante la posibilidad de que alguna de las acciones contenidas cree alguna variable temporal.

- En todos los objetos de clase *BoolExpr*, añadiendo además la expresión *self*.

[1] Concretamente para el preprocesado, solo importa su tipo
[2] O eventos, acciones o parámetros contenidos por este widget

Este sistema de tablas es usado cada vez que se instancia una variable temporal en las acciones de creación de entidades, añadiendo a la última tabla de la lista el nombre y el tipo de la variable temporal que instanciará.

Así pues, al preprocesar subexpresiones ha de buscarse en esta lista de tablas. Si se encuentra, ha de declararse entonces en EOS una nueva variable cuyo tipo será el obtenido de las tablas y sustituir en la expresión OCL la subexpresión tratada con la variable EOS declarada. De no encontrarse en las tablas la subexpresión, la expresión OCL será inválida.

Traducción de OCL estándar a SQL

Aunque se diseñe como un lenguaje sencillo y con una especificación completa, OCL es un lenguaje bastante complejo en comparación a las subexpresiones GUI y de seguridad. Por ello se delega la traducción de OCL estándar a SQL a la librería MySQL4OCL. Dada una expresión OCL, un fichero, los tipos de las variables globales y una base de datos como la explicada en la sección anterior, esta librería traduce la expresión a un procedimiento MySQL que guarda en el fichero y devuelve la llamada SQL que ha de realizarse. Esta llamada es una cadena de texto de la forma:

```
call id(lista de variables necesarias)
```

Por último, se sustituye el valor de todos los atributos que tenían como valor una expresión OCL en el modelo de entrada SecumlAndGui por la cadena devuelta por MySQL4OCL. Esta llamada al método tiene como argumentos las variables EOS generadas al preprocesar la expresión. Para recuperar el significado de estas variables, el módulo generador de código vuelve sobre sus pasos y sustituye estas variables EOS con la subexpresión de la que provienen.

De esta manera desaparecen del modelo todas las expresiones OCL, siendo sustituidas por cadenas de texto que permiten hacer llamadas a métodos SQL tras sustituir las subexpresiones GUI y de seguridad por los valores concretos que se conocerán en tiempo de ejecución.

Implementación

El módulo generador de código implementa el algoritmo infomalmente explicado anteriormente haciendo uso de las clases que pueden verse en la figura 3.1, cuyas clases se explican a continuación:

- **VaadinCodeGen**: Es el punto de entrada del módulo generador de código. Cuando el atributo *generateSQL* está activado, una llamada al método *execute()* traducirá las expresiones OCL, delegando la tarea en *SsgToSql*.

- **SsgToSql**: Es el punto de entrada de la traducción de expresiones a SQL. Se encarga de iniciar correctamente EOS.
- **IContext**: Representa una tabla, que en este caso relaciona subexpresiones con su tipo.
- **HierarchicalContext**: Se trata de un *IContext* que tiene otro *IContext* asociado. En caso de la primera tabla no contenga la subexpresión que se intenta resolver, delega la resolución en la segunda.
- **Scope**: Usando un *IContext* raiz y encadenando el número necesario de *HierarchicalContext* implementa la lista de tablas con la que es capaz de resolver las subexpresiones.
- **ModelTransformation**: Recorre el modelo SecumlAndGui y utiliza los métodos de *Scope* para apilar y desapilar tablas y resolver expresiones.
- **Expression**: Envuelve un *String* con métodos que permiten encontrar subexpresiones OCL y sustituirlas por otras cadenas de texto de manera eficiente.

3.3. Generación de CSS

El módulo generador de código de SmartGUI Viewer recibe un modelo XYLayout de SmartGUI y lo transforma en código entendible por SmartGUI Viewer. Aunque en el lado del servidor los componentes Vaadin son objetos Java, en el lado del cliente se implementan con HTML. El framework clasifica cada componente HTML con una clase CSS concreta.

La transformación de XYLayout a CSS es sencilla. Basta con establecer el atributo CSS *position* al valor *absolute* y los atributos *top*, *left*, *width* y *height* al valor que tienen en el modelo, fijando las unidades en píxeles.

Ciertos widgets necesitan hojas de estilo más complejas, por ejemplo, las tablas requieren definir reglas para la cabecera, el cuerpo y la caja que contiene a ambos.

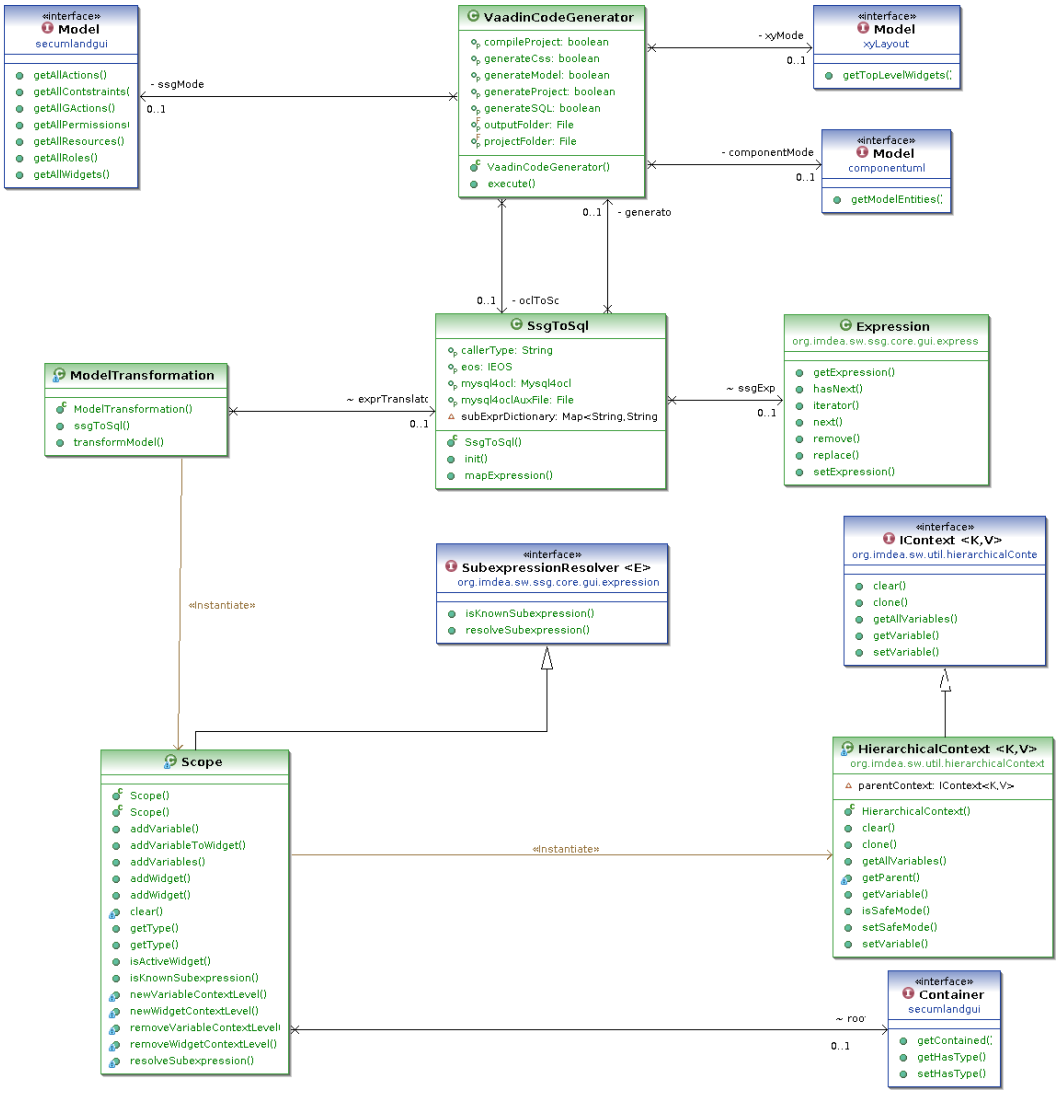

Figura 3.1: Módulo procesador de expresiones

Capítulo 4

Motor web

Resumen: Este capítulo describe SmartGUI Viewer, el programa que, usando los métodos SQL y las hojas de estilo, es capaz de interpretar el modelo SecumlAndGui en forma de aplicación web sobre el framework Vaadin. Primero se expondrá el diseño del código, así como algunas peculiaridades Vaadin. A continuación se mostrará el ciclo de vida de los widgets, concrétamente cómo se calcula si se tiene o no permiso para disparar sus eventos y cómo se notifican entre ellos los cambios de los valores de sus variables, lo cual permite mantener la interfaz sincronizada. Finalmente se explicará cómo se realizan las llamadas a los métodos SQL, en especial cómo se resuelven las subexpresiones GUI y de seguridad.

4.1. Diseño del código

SmartGUI Viewer es una aplicación web Java que se divide en cuatro partes:

- **Widgets**: Contiene las clases que conectan los widgets de SecumlAndGui con los componentes Vaadin. Para facilitar la distincion entre las clases de los widgets del modelo (*Label*, *Button*, *Window*...), los de este paquete añaden una "S" al widget del que provienen (*SLabel*, *SButton*, *SWindow*...). Además existe un STable que se diferencia de SComboBox.

- **Exceptions**: Contiene las clases que se lanzarán como excepción durante la ejecución de la aplicación.

- **Wrappers**: Contiene una serie de clases que hacen de envoltorio a los posibles valores de las variables y los resultados de los métodos

SQL. Existe un envoltorio para enteros, uno para booleanos, otro para cadenas de texto, para listas... Todos los envoltorios implementan la interfaz *DataWrapper*.

- **Core**: Contiene el núcleo lógico de la aplicación. Las clases principales son las siguientes:
  - **SsgServlet**: Es la clase que responderá a las peticiones de los clientes. En Vaadin las peticiones originadas por usuarios desconocidos generan una instancia de com.vaadin.Application, a la que se les delegará en las sucesivas peticiones. En el caso de *SsgServlet* la aplicación lanzará una instancia de *SsgApplication*. Todos los usuarios compartirán una única copia de *SsgServlet*, por lo que se utiliza para almacenar el modelo SecumlAndGui que contiene la información de la aplicación a ejecutar y otros atributos de configuración como la ruta, usuario y contraseña usadas para acceder a la base de datos.
  - **SsgApplication**: Esta clase puede verse como el punto de entrada de la aplicación para un cierto usuario siendo el método *init* equivalente al método *main* en una aplicación de escritorio Java. En Vaadin todos los componentes que se presentan al usuario están asociados a una aplicación y por tanto puede servir para, dado un componente, obtener información de la sesión del usuario. Además esta clase almacena un historial de las ventanas abiertas y posee métodos tanto para moverse por dicho historial y como para abrir nuevas ventanas.
  - **Session**: Cada *SsgApplication* está asociado con un objeto de *Session*. Esta clase implementa, usando los mismos objetos *IContext* que en la generación de código, una pila de ámbitos que será usada durante la ejecución de acciones para almacenar las variables temporales. Además se asocia con un objeto *SsgDAO*, al cual se delega la implementación de las acciones sobre el modelo, y otro SmartGUIExpressionResolver, al cual se delega la ejecución de evaluación de las expresiones, que serán siempre llamadas a métodos SQL. Por último dispone también de una instancia de *ActionExecutor*, clase que implementa la ejecución de métodos y acciones.

4.2. Ciclo de vida de un widget

Es habitual describir el funcionamiento de los componentes software explicando su ciclo de vida. El de un *SWidget* puede verse en la figura 4.1

Por construcción se garantiza que dado un contenedor, todos sus widgets estarán construídos (estarán instanciados como objetos Java) antes de

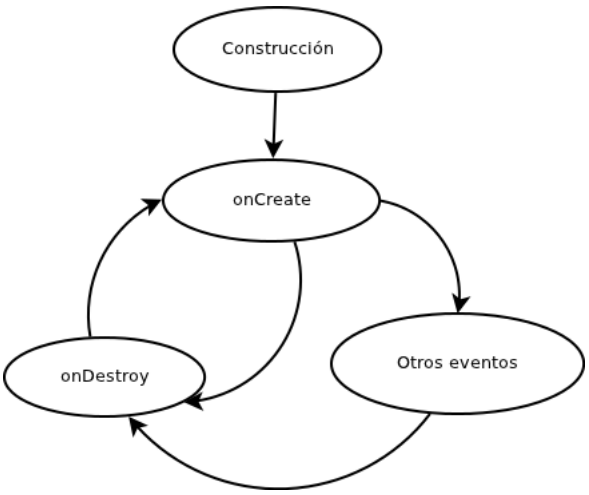

Figura 4.1: Ciclo de vida de un *SWidget*

que ninguno de ellos reciba el primer evento *onCreate*. Además, todos ellos estarán en estado *onCreate* antes de que reciban eventos de cualquier otro tipo. Durante el evento *onCreate* de cada contenedor se instancian los widgets que este contiene. En el caso de las ventanas es un proceso sencillo: para cada widget del modelo se genera el *SWidget* correspondiente. Instanciar el contenido de las clases *STable* y *SComboBox* es más complejo.

Al modelar una tabla[1], a diferencia de lo que ocurre con las ventanas, los widgets con los que se asocia no son los que la tabla contendrá, sino que hacen las veces de plantilla para una cierta columna. La tabla tendrá un número de filas que se conocerá en ejecución y para cada celda se instanciará un widget distinto. Así, el proceso de instanciación de los widgets de una tabla es el siguiente:

1. Evaluar la expresión asociada a su filtro. El resultado es una lista de entidades del modelo de datos.
2. Para cada entidad de la lista anterior se genera una fila.
3. Para cada widget contenido en el modelo se genera una columna.
4. Para cada celda de genera un *SWidget* usando como entrada el widget columna.
5. Cada *SWidget* se memoriza la entidad que le corresponde por fila. Esta asociación será consultada cuando se haga uso de la variable *row* de la tabla.

[1]Puesto que un *SComboBox* puede verse como una versión simplificada de un *STable*, únicamente se explicará en detalle cómo se instancia el contenido de una tabla.

Después de instanciar los widgets contenidos, se dispara el evento *onCreate* para cada uno de ellos.

Seguridad

Antes de que un widget dispare un evento, SmartGUI Viewer comprueba que el usuario tenga permiso para ejecutar todas las acciones asociadas a él. Para ello se utilizan las restricciones de seguridad inferidas por SmartGUI. Estas restricciones, que dependen del rol del usuario de la aplicación, son objetos de clase *secumlandgui.Formula*, es decir, conjunciones o disjunciones de otras fórmulas y fórmulas atómicas que se evalúan como un método SQL.

Un widget estará activado (es decir, interactuará con el usuario) si y solo si el usuario tiene permiso para disparar el evento *onCreate*. Cuando se evalúa este permiso, puede ocurrir uno de estos tres resultados:

1. La comprobación tiene éxito y el resultado indica que el usuario puede disparar el evento.

2. La comprobación tiene éxito y el resultado indica que el usuario no puede disparar el evento.

3. La comprobación no tiene éxito. Esto puede ocurrir cuando para realizar alguna de las llamadas a métodos se requiere una variable de widget que aun no tiene valor[2].

Para mantener la coherencia entre los widgets, es necesario, por tanto, que cuando se modifique el valor de una variable de widget, el cambio sea notificado a los widgets de los que depende, para que estos reevaluen sus permisos y sus propias variables. Sin embargo la relación entre un widget y las variables de las que depende no es conocida por SmartGUI Viewer, por lo que recurre a avisar a todos los widgets, que por norma general vuelven a ejecutar su evento *onCreate*.

Este sistema, aunque correcto, introduce una alta carga computacional innecesaria. Por ejemplo, las tablas y los comboboxes no recalculan sus filas al cambiar las variables de las que dependen debido a que, con esta implementación, también habrían de recalcularlas al cambiar variables de las que no dependen, lo cual supone un coste computacional excesivamente alto. Esto obliga al modelador a introducir una acción *Reload* que indique explícitamente que ha de recalcularse la ventana siempre que quiera recalcular el contenido de estos widgets.

[2]Esto sucede debido a que SmartGUI no define el valor que tienen las variables hasta que éstas no se fijan explícitamente y, por otro lado, tampoco indica si un widget depende o no de los valores de otros, por lo que SmartGUI Viewer llama al evento *onCreate* sobre ellos en un orden arbitrario

4.3. Llamadas a métodos

Como se explicó en el capítulo dedicado a la generación de código, en el modelo SecumlAndGui que utiliza SmartGUI Viewer, las expresiones OCL extendidas son sustituidas por las llamadas a los métodos SQL que implementan dichas expresiones. Estos métodos pueden requerir ciertos argumentos que serán subexpresiones GUI o de seguridad. Así como para generar el código de las expresiones OCL hace falta conocer el tipo de las subexpresiones que contenía, para ejecutarlos hace falta resolver estas subexpresiones de manera que sus valores sean usados como argumentos del método SQL correspondiente.

La clase *MySqlSsgDAO*, que implementa las interfaces *SsgDAO* y *ExpressionResolver*, es la encargada de realizar toda comunicación con la base de datos y, en concreto, de analizar las llamadas a los métodos para resolver las subexpresiones que contenga. Las subexpresiones GUI y de seguridad pueden ser las mismas que se contemplan a la hora de generar código:

- **self**: Para referirse, en una restricción de un permiso, al recurso al que se quiere acceder.
- **caller**: Para referirse, en una restricción de un permiso, al actor que intenta acceder al recurso.
- **variable temporal**: Accesible solo desde acciones y sus parámetros, habiendo sida declarada por alguna acción que preceda a la acción que la usa.
- **variable de widget**: Accesible desde cualquier widget (y sus acciones asociadas) siempre y cuando comparta ventana con el widget que posee la variable. Además existen ciertas restricciones como las impuestas por la propia naturaleza de las tablas y comboboxes.

Resolución de subexpresiones

La resolución de estas subexpresiones es diferente según su tipo. Las subexpresiones *caller*, *self* y las variables temporales se delegan a la sesión, por lo que, si esta se gestiona correctamente al ejecutar las acciones, resolverlas es tan sencillo como consultar sucesivamente las diferentes tablas que forman el ámbito o contexto de la sesión, de manera similar a lo explicado al generar el código. La gestión de la sesión consiste en añadir nuevas tablas a la sesión en las siguientes situaciones: A esta lista de tablas ha de añadirse niveles en los siguientes casos:

- En todos los eventos y todas las acciones condicionales.
- En todos los objetos de clase *BoolExpr*, añadiendo además la expresión *self*.

Ahora no es necesario añadir un nuevo nivel ni en las tablas ni en los comboboxes, puesto que la variable *row* ya se gestiona de manera tal que solo será resoluble si el widget que la usa está contenido por la tabla o el combobox.

Las variables de widget, sin embargo, resultan más interesantes. A la hora de generar el código SQL correspondiente a una expresión OCL, solo era necesario conocer el tipo de la variable, algo que puede hacerse con un análisis estático de manera tan sencilla como navegar por el modelo tal como indique su identificador global y obtener el tipo del objeto *WidgetVariable*. En el momento de ejecutar un método, sin embargo, es necesario conocer el valor de la subexpresión. Esto no puede hacerse recorriendo la cadena de widgets indicada por el id de los mismos puesto que en las tablas y los comboboxes por cada objeto de clase *Widget* en el modelo pueden existir varios objetos *SWidget* en ejecución. Todos estos *SWidgets* comparten id, pero cada uno de ellos tiene sus propias instancias de variables y por tanto sus propios valores. Por ello el id del widget no resulta unívoco y, por tanto, no puede realizarse la navegación.

Las peticiones que recibe *MySqlSsgDAO* para ejecutar un método están siempre asociadas a un widget. Si el método a ejecutar contiene una subexpresión de tipo variable de widget, se delega en el widget asociado a la petición el resolver esta subexpresión.

Para resolver una variable, los widgets implementan el método *resolveWidgetVariable*, que requiere los tres siguientes parámetros:

- **widgetChain**: Una cadena de ids de widget, que representa la navegación desde una ventana hasta el widget que contiene la variable a resolver, de tal manera que el widget correspondiente al id de la posición i contiene en el modelo un widget de cuyo id es el de la posición $i + 1$.

- **varId**: El id de la variable a resolver.

- **petitioners**: Una pila que almacena los widgets que realizan la petición. Esta pila es usada para resolver ciertas navegaciones no triviales como las de los comboboxes y tablas.

Así, cuando se ha de resolver una variable de widget, se llama al método *resolveWidgetVariable* del widget asociado a la expresión. Los argumentos *widgetChain* y *varId* se obtienen de la subexpresión y como *petitioners* se envía una pila con el propio widget.

En general, los widgets resuelven las peticiones de resolución de variable delegándolas en el widget que les contiene, añadiendose a sí mismo en la cima de *petitioners*. Repitiendo este proceso suficientes veces, la petición llegará a la ventana que contiene al widget que ha demanda resolver la variable. A

partir de entonces la petición baja a través de los contenedores de la siguiente manera:

```
Context SContainer::
resolveWidgetVariableDescendig(
        petitioners : Sequence(SWidget),
        widgetChain : Sequence(String),
        varId: String,
        index : Integer) : DataWrapper
    if self.canResolve(petitioners, widgetChain, varId, index) then
        result := self.resolve(petitioners, widgetChain, varId, index)
    else
        var sw : SWidget := getDescendingWidget(
            petitioners,
            widgetChain.at(index)
        )
        if sw.isOclUndefined() then
            error
        else
            result := sw.resolveWidgetVariableDescendig(
                petitioners->removeLast(),
                widgetChain,
                varId,
                index + 1
            )
        endif
    endif
```

Cuando SWidget::canResolve devuelve cierto, entonces SWidget::resolve ha de devolver el valor de la variable. La especificación de SWidget::canResolve es de la siguiente:

```
Context SWidget::
carResolve(
        petitioners : Sequence(SWidget),
        widgetChain : Sequence(String),
        varId: String,
        index : Integer) : Boolean

    result :=
        petitioners->size() = index
        and self.id = petitioners->at(index - 1)
        and self.variables.id->contains(varId)
```

Más interesante es la función SContainer::getDescendingWidget, cuya implementación dependerá de la naturaleza de cada contenedor:

- **SWindow**: En las ventanas se tiene un atributo que relaciona identificadores con los SWidgets que contiene, por lo que basta con consultar ese atributo para obtener el siguiente widget en la cadena.
- **STable**: Dada una columna, todos los widgets de esa columna son idénticos al widget con el que se modela la columna, por lo que comparten id y esto los hace imposibles de distinguir externamente. Por ello una subexpresión que intente acceder a un widget *w1* contenido en una tabla solo podrá evaluarse si quien ha hecho la peticion (es decir, quien está en la cima de la pila *petitioners*) concuerda con la variable elegida. Esto es:

```
Context STable::getDescendingWidget(
        petitioners : Sequence(SWidget),
        id : String) : SWidget
    var sw : SWidget := petitioners.peekLast()
    if(self.containedWidgets->contains(sw) and sw.id = id)
        result := sw
    else
        error
    endif
```

De esta manera cualquier subexpresión, siempre y cuando sea correcta, podrá ser resuelta.

Capítulo 5

Trabajo futuro

Resumen: Aunque se trate de una herramienta prometedora SmartGUI se encuentra todavía en un estado poco maduro en el que existen multitud de ramas de trabajo. En este capítulo se enumeran alguna de las carencias de la herramienta.

Modularidad

La herramienta se encuentra actualmente dividida en módulos. Por un lado están las transformaciones de modelo a modelo, por otro la generación de código y por otro el motor web. Sin embargo sería deseable que las dependencias entre estos módulos sea tan pequeña que, de disponer por ejemplo de otro módulo generador de código, el cambio entre uno y otro fuera sencillo. Actualmente esto no ocurre, siendo necesario recodificar los módulos que dependen del módulo cambiado.

Representación gráfica

Como se explicó en el capítulo correspondiente, SmartGUI no define cómo se presentan los componentes gráficos. De manera temporal, la distribución de los widgets en la pantalla se extrae de cómo el usuario los dibuja en la herramienta. Esto es poco preciso, pues las herramientas usadas no están diseñadas para ello y además es poco expresivo, no pudiendo definirse atributos tan comunes en las interfaces gráficas como el color de un widget, tipos de letras o usar encuadrar los widgets en layouts que faciliten el diseño y lo hagan escalable a distintas resoluciones.

Grafo de dependencias

En el capítulo anterior se describió la costosa estrategia usada por SmartGUI Viewer para informar a los widgets cuando uno de ellos cambia el valor de alguna variable. Para implementar una estrategia más eficaz es necesario disponer de una estructura de datos que, dada una variable, indique que widgets han de ser notificados de su cambio.

Concurrencia

Puesto que Vaadin lanza una aplicación por cada usuario, SmartGUI Viewer es a nivel gráfico, concurrente. Sin embargo el manejo de la base de datos no lo es. Por un lado los métodos generados por MySQL4OCL almacenan datos temporales que suponen una sección crítica que no está protegida. Por otro, las acciones sobre modelos presuponen que la base de datos no ha cambiado de estado desde la última lectura. Por ejemplo, al modificar el atributo de una entidad, no se comprueba que otro usuario no haya destruido esa entidad instantes antes.

Seguridad

El algoritmo de seguridad de SmartGUI consiste en chequear antes de ejecutar un evento, si con el estado actual de la aplicación (considerando como estado todas las variables de la ventana y el estado de la base de datos) el usuario tiene permiso para ejecutar todas las acciones del evento. Sin embargo durante la ejecución del evento ciertas acciones pueden cambiar el estado de la aplicación, haciéndolo transitar a uno en el cual el usuario podría no tener permiso para ejecutar las acciones siguientes. Esto puede deberse o bien a un error de modelado (el cual una herramienta madura debería detectar y avisar al modelador) o bien a un acceso concurrente que modifique la base de datos.

Por otro lado la seguridad de acceso se comprueba en las acciones sobre el modelo de datos pero no en las expresiones OCL. Un ejemplo de esto es la acción *set*, que mediante fija el valor de una expresión en una variable de widget. A las navegaciones por el modelo de datos que realice esta expresión no se le aplica la política de seguridad, por lo que un diseño descuidado del modelo ActionGUI hace inútil la política de seguridad.

Expresividad

Colecciones

Las expresiones OCL que acepta MySQL4OCL, de momento, no admiten variables de tipo colección, lo cual acota significativamente la expresividad del lenguaje.

Acciones definidas por el usuario

Las acciones que define SmartGUI se centran en el modelo de datos y en el flujo de información en la interfaz gráfica. Sin embargo no permite realizar acciones externas, como por ejemplo enviar un correo electrónico, abrir una página web o acceder a un archivo. Estas acciones podrían añadirse una a una a SmartGUI, aunque sería preferible permitir al usuario definir las acciones que necesita y las restricciones de acceso a ellas, permitiendo así que SmartGUI se adapte a las necesidades de cada aplicación.

Widgets definidos por el usuario

Al igual que ocurre con las acciones, SmartGUI define una serie de widgets que pueden resultar escasos para ciertas aplicaciones. Una estructura modular podría permitir al usuario crear sus propios widgets y añadirlos a la colección soportada por SmartGUI.

Apéndice A

Autorización de difusión

Los abajo firmantes autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Gonzalo Ortiz Jaureguizar

Apéndice B

Lista de palabras

MDA, seguridad, generación, automática, interfaces, gráficas, ActionGUI, RBAC

Bibliografía

Basin, D., Doser, J. y Lodderstedt, T. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, vol. 15(1), páginas 39–91, 2006.

Care Technologies. Olivanova – the programming machine. 2011. `http://www.care-t.com`.

Clavel, M., Egea, M. y de Dios, M. A. G. Building an efficient component for ocl evaluation. *ECEASST*, vol. 15, 2008.

Eclipse Model to Model (M2M) Project. The operational QVT transformation engine. `http://www.eclipse.org/modeling/m2m/`, 2011.

Egea, M., Dania, C. y Clavel, M. Mysql4ocl: A stored procedure-based mysql code generator for ocl. *ECEASST*, vol. 36, 2010.

Ferraiolo, D. F., Sandhu, R. S., Gavrila, S., Kuhn, D. R. y Chandramouli, R. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, vol. 4(3), páginas 224–274, 2001.

Microsoft. Visual studio lightswitch. 2010. `http://www.microsoft.com/visualstudio/en-us/lightswitch`.

Object Management Group. Object constraint language specification version 1.1. 1997. OMG Document available at `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.

Object Management Group. Model driven architecture guide v. 1.0.1. Informe técnico, OMG, 2003. OMG document available at `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

Object Management Group. MOF-Queries, Views and transformations (QVT)-final adopted specification. Informe técnico, OMG, 2005. `www.omg.org/docs/ptc/05-11-01.pdf`.

Web Models Company. Web ratio – you think, you get. 2010. `http://www.webratio.com`.